# Multiple Alignment with Hidden Markov Models

Kalin Vetsigian

November 15, 2001

Multiple alignment of sequences is an important problem in bioinformatics. For example, multiple alignment of proteins belonging to the same family can provide valuable information for the common protein structure. Looking at a multiple alignment one can identify patterns which are not obvious if one looks at just pairwise alignments between members of the family. Traditional alignment schemes first choose a scoring matrix and gap penalties and then use dynamic programming which gives exact result in $O(N^2)$ operations for two sequences of length N. For multiple alignment of $K$ sequences dynamic programming requires $O(N^K)$ operations, which is exponential in $K$. Therefore the algorithm is not feasible for multiple alignment of more than a few sequences. On the other hand, protein families often contain hundreds of sequences. To avoid the problem various heuristic algorithms were introduced.

However, it is often the case that the solution to a multiple alignment problem depends strongly on the scoring scheme. This poses the question of how to choose the most biologically relevant one. Another very important issue is that the scoring matrices and gap penalties are assumed to be independent of position along the sequence, while gene families often exhibit regions which are highly conserved and regions which are highly variable. In order to obtain the most biologically relevant multiple alignment these differences should be taken into account. If we want to construct multiple alignments based only on primary structure information, the variations of scoring matrices and gap penalties should be deduced from the sequences themselves. Hidden Markov Models (HMMs) are an implementation of the idea that the scoring parameters should guide the multiple alignment as much as the alignment should determine the parameters.

A Hidden Markov Model (HMM), $\lambda$, is a stochastic machine for generating (amino acid) sequences. Different sequences are generated with different probability. A sequence $S$ is generated with probability $P(S|\lambda)$. The goal is: Given an universe, $\Lambda$, of HMMs to chose the member, $\lambda^*$, which maximizes $\prod_{k=1..K} P(S_k|\lambda^*)$, where $S_1, ..., S_k$ are the sequences we try to align. The procedure of choosing $\lambda^*$ can be thought of as *training* a HMM using training set $\{S_1, ..., S_K\}$. This is analogous to neural networks. The hope is that, after the training, the model have captured the essence of "what it means to be a member of this particular protein family". If so, $\lambda^*$ will preferably generate amino acid sequences which belong to the family, i.e. sequences that exhibit the same

structure as all the members of the family. Reversely, one can take a sequence $S$ and decide whether it belongs to the family based on the value of $P(S|\lambda^*)$. As always when one does training (or fitting), the success depends, among other things, on the relative sizes of the universe $\Lambda$ (number of adjustable parameters) and the training data set (number of sequences). The architecture of the HMMs in $\Lambda$ should be carefully chosen so that it manages to capture the training data patterns with as few parameters as possible.

Mathematically, a HMM is defined by specifying: a set of states $s = \{s_1, ..., s_n\}$, an alphabet $v = \{v_1, ...v_M\}$, transition probabilities between states $A = \{a_{ij}\}, 1 \leq i, j \leq n$ and a set of probability distributions $B = \{b_j(o)\}$, where $j$ specifies a state, and $o$ - a letter from the alphabet. The system evolves in discrete steps. At each step, $t$, the system is in one of the states in $s$, say $s_i$. A letter from the alphabet is generated according to $b_i(o)$, then a new state is chosen according to the set of probabilities $a_{ij}, j = 1..n$ and the cycle continues. A HMM can be represented by a graph in which the nodes corresponds to the states, and arrows connect nodes for which there is a non-zero transition probability.

The concept of HMM described above is very general, and different graph topologies are used in different applications. For the purposes of multiple alignment of sequences Krogh et al. [1] proposed the architecture shown on Figure 1. The alphabet, $v$, is the set of 20 amino acids. There are three types of states (not counting the begin and end states): match (rectangles), insertion (diamond) and delete (circle). The delete states are special in that they don't generate amino acids. To understand the relevance of this architecture imagine a family of proteins with different sequences which have a similar 3D structure. While there are very many sequences capable of creating this structure, the structure imposes severe constraints on the sequences. For example: the structure might start with an $\alpha$-helix about 30 aa long, followed by a group that binds to TT dimers, followed by about 20 aa with hydrophobic residues, etc. Basically, we walk along the sequence and enter into different regions in which the probabilities to have different amino acids are different (for example, it is very unlikely for members of the family to have an amino acid with hydrophilic residue in the hydrophobic region, or gly and pro are very likely to be present at sharp bends of the polypeptide chain, etc.). Different columns in the graph, called modules, correspond to different positions in the 3D structure. Each position has its own probability distribution, $b_{m_i}$, giving the probabilities for different amino acids to occur at that position. Each position can be skipped by some members of the family. This is accounted for by the delete states. There might also be members of the family that have additional amino acids relative to the consensus structure. This is allowed by the insertion states.

This type of description is far from complete for it fails to capture the interactions and correlations between amino acids at different positions along the chain. Even correlations between adjacent amino acids are ignored. In addition, implicit in the architecture is an exponential distribution of insertion lengths. This is because there is a constant probability for transition from an insertion state to itself. Some of this problems can be solved by generalizing the concept of state in a HMM. For example, we can allow for states that generate not one

but $l$ letters, where $l$ is drawn from a state specific probability distribution, and the $l$ letters are drawn from a $l-$dimensional joint distribution.

The number of modules is chosen in advance and the universe $\Lambda$ will consist of HMMs with all possible values of $B$ and $A$ but a fixed graph topology. This constraint can be relaxed by allowing a dynamical adjustment of the module number during training. If during training the parameters of the HMM stabilize in such a way that within a given module the delete state is more likely to be visited than the match state then the corresponding module is removed. Similarly, if certain insertion states are visited more often than the corresponding match states, additional modules should be added at that position.

The sequence of states visited during a single run of the HMM is called a *path through the model*. A given sequence of amino acids can be generated by different paths through the model. Each one is said to represent *alignment of the sequence to the model*. Given paths through the model for each of the sequences $S_1, ..., S_K$ one can write down the corresponding multiple alignment in the usual notation. See Figure 2 for an example. If for each sequence we take the most likely path through the model then we get the optimal multiple alignment.

The usefulness of HMMs comes from the existence of efficient algorithms to: (1) score a sequence $S$ given a model $\lambda$, i.e. compute $P(S|\lambda)$, (2) Find an optimal alignment of a sequence to a model, i.e. find a sequence $Q$ of states that maximizes $P(Q|S, \lambda)$, (3) perform the training procedure. The total number of operations required for multiple alignment is just $O(KN^2)$, which is linear in the number of sequences! In the remaining of the paper these algorithms will be described in some detail following [2]. The delete states, which are special for not generating output, will be ignored for simplicity.

Let $S = o_1, o_2, ..., o_T$ be a sequence of amino acids. Let $q_t$ denote the state at time $t$, with $q_0 = s_0 \equiv$ BEGIN and $q_{T+1} = s_{n+1} \equiv$ END. Define the *forward variable* $\alpha_t(i) \equiv P(o_1 o_2 ... o_t, q_t = s_i|\lambda)$. $\alpha_t(i)$ is the probability to observe the (partial) sequence $o_1 o_2 ... o_t$ up to time $t \leq T$ and state $s_i$ at time $t$ given the model $\lambda$. $\alpha_t(i)$ can be computed recursively by using

$$\alpha_{t+1}(i) = b_i(o_{t+1}) \sum_{j=1}^{n} \alpha_t(j) a_{ji} \tag{1}$$

with initial condition $\alpha_1(i) = a_{oi} b_i(o_1)$ Then the probability to observe $S$ given $\lambda$ is $P(S|\lambda) = \sum_{i=1}^{n} \alpha_T(i) a_{i,n+1}$. This formulas are written in a notation appropriate for a general HMM. For the specific topology considered most of the $a_{ij}$'s are zero which speeds up the computation relative to the general case. For later use define the *backward variable* $\beta_t(i) = P(o_{t+1} o_{t+2} ... o_T|q_t = s_i, \lambda)$, giving the probability for partial observation sequence from $t+1$ to the end, given that the state at $t$ is $s_i$. Again, this can be computed inductively using the formula

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j), \tag{2}$$

with initial condition $\beta_T(i) = a_{i,n+1}$.

The algorithm for finding the most likely path through the model, called *Viterbi* algorithm is also inductive. The goal is to find the path $Q^* = \{q_1^*, ..., q_T^*\}$ for which $P(Q|S,\lambda)$ is maximized. Because of $P(Q|S,\lambda) = P(Q,S|\lambda)/P(S|\lambda)$, this is equivalent to maximizing $P(Q,S|\lambda)$. We compute

$$\delta_t(i) = \max_{q_1,...,q_{t-1}} P(q_1, ..., q_{t-1}, q_t = s_i|\lambda), \tag{3}$$

starting with $\delta_1(i) = a_{0i}b_i(o_1)$, and using the recursion relation $\delta_{t+1}(j) = b_j(o_{t+1}) \max_i[\delta_t(i)a_{ij}]$. Then the probability at the maximum is $P^* = P(Q^*, S|\lambda) = \max_i[\delta_T(i)a_{i,n+1}]$. To recover $Q^*$ we must also keep track of the state $i$ that yielded $\delta_{t+1}$. Therefore we store the array $\psi_{t+1}(j) = \text{argmax}_i[\delta_t(i)a_{ij}]$ at each inductive step. Then we recover $Q^*$ by tracing back using $q_t^* = \psi_{t+1}(q_{t+1}^*)$, and starting with $q_T^* = \text{argmax}_i[\delta_T(i)a_{i,n+1}]$.

The idea behind the training algorithm is to update $a_{ij}$ and $b_i(o)$ using

$$a_{ij}^{\text{new}} = \frac{\text{expected num. of trans. from } s_i \text{ to } s_j \text{ given } \{S_k\} \text{ and } \lambda^{\text{old}}}{\text{expected num. of transitions from } s_i \text{ given } \{S_k\} \text{ and } \lambda^{\text{old}}} \tag{4}$$

$$b_i(o)^{\text{new}} = \frac{\text{expected num. of times in } s_i \text{ AND symbol } o \text{ generated given } \{S_k\} \text{ and } \lambda^{\text{old}}}{\text{expected number of times in } s_i \text{ given } \{S_k\} \text{ and } \lambda^{\text{old}}}.$$

To compute these expected values define $\xi_t^{(k)}(i,j) = P(q_t = s_i, q_{t+1} = s_j|S_k, \lambda)$ which can be expressed in terms of the forward and backward variables for each sequence:

$$\xi_t^{(k)}(i,j) = \frac{\alpha_t^{(k)}(i)a_{ij}b_j(o_{t+1})\beta_{t+1}^{(k)}(j)}{P(S_k|\lambda)}. \tag{5}$$

Then $\gamma_t^{(k)}(i) = \sum_j \xi_t^{(k)}(i,j)$ is the probability to be in state $s_i$ given a sequence $S_k$ and $\lambda$. Now it easy to see that

$$a_{ij}^{\text{new}} = \frac{\sum_{k=1}^K P(S_k|\lambda)^{-1} \sum_{t=1}^{T_k-1} \xi_t^{(k)}(i,j)}{\sum_{k=1}^K P(S_k|\lambda)^{-1} \sum_{t=1}^{T_k-1} \gamma_t^{(k)}(i)} \tag{6}$$

$$b_i(o)^{\text{new}} = \frac{\sum_{k=1}^K P(S_k|\lambda)^{-1} \sum_{t=1}^{T} \gamma_t^{(k)}(i)\delta_{o_t,o}}{\sum_{k=1}^K P(S_k|\lambda)^{-1} \sum_{t=1}^{T} \gamma_t^{(k)}(i)}$$

This training procedure is guaranteed to converge to a local maximum in the parameter space. The main problem is that in most applications of interest the optimization surface is very complex and there are very many local maxima which trap the model and prevent it from reaching a global maximum. A traditional method to fight with this problem is to repeat the training procedure with different randomly chosen initial parameters, and then select the best local maximum found.

# References

[1] Krogh, A., Brown, M., Mian S., Sjolander, K. & Haussler, D., Hidden Markov Models in computational biology: Applications to protein modeling. *J. of Mol. Biol.* **235**, 1501-1531 (1994)

[2] Rabiner, L., A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proc. IEEE* **77**, 257-285 (1989)